SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 18(5), 485–491 (MAY 1988)

Short Communication

A MICRO-KERNEL FOR CONCURRENCY IN C

GORDON V. CORMACK

Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

SUMMARY

A micro-kernel that supports concurrent execution of C procedures within a single user process is described. A micro-kernel provides only four primitives, which have been used to build a number of higher-level abstractions, including support for distributed processing. The micro-kernel differs from other efforts in that it is small and efficient, it is written entirely as a non-privileged user program, and it provides fine-grained unpredictable interleaving of execution.

KEY WORDS Concurrency C Synchronization Task Operating system

INTRODUCTION

In this communication, we describe the implementation and use of concurrent programming primitives that can be called from a C program.[1] This implementation was undertaken to meet our need for a laboratory tool to study concurrent algorithms, notations for concurrent programming, and distributed programming. This tool was needed as an educational device, to provide run-time support for concurrent languages, and as a means to simulate other systems being developed. All of the existing tools for concurrency that we investigated have excessive baggage: either they have cumbersome and restrictive notations, or they entail high execution overhead, or they do not provide true concurrency.*

Unix[2, 3] processes, like those in most operating systems, are unsuitable for a number of reasons. Unix processes run in separate address spaces — this property is no doubt desirable in a large number of cases, but it did not meet our need to be able to test low-level synchronization and communication protocols. Furthermore, a large class of notations for concurrent programming is not easily mapped to such a process model. Because there is no shared memory, the programmer is forced to use the synchronization and communication primitives provided by the system, namely *pipes*. Although these facilities provide communication, there is no easy way that they can be used to provide synchronization such as locking. A further drawback of having separate address spaces is that a great deal of overhead is incurred in creating a process, and in performing a context switch. A final limitation of Unix processes is that there is a system-imposed limit on the number of processes that a user may create — we have applications that exceed this limit by orders of magnitude.

Using a concurrent programming language carries with it its own sort of baggage: all programming must be done in the language; the language imposes a particular model of concurrency and a particular set of primitives; and there must be an *implementation* of the language with appropriate run-time support. For example, PL/I[4] has provision to execute a procedure concurrently (call task), but has no synchronization facility capable of implementing locking. Ada[5] has extensive provision for tasking, but imposes a particular methodology. Concurrent C[6] imposes a special compiler and Ada-like primitives on the programmer. Concurrent Euclid[7] provides concurrency only between statically declared process (dynamic process creation is impossible) and provides synchronization only through monitors.

We found that it was much less effort to implement a concurrent programming facility from scratch. It is well known, in general, how to multiplex the execution of many processes on a single processor, but programs that do so are typically confined to the innermost workings of operating systems and rely on privileged commands to ensure non-preemption. Our micro-kernel was written to run as a user program under the Unix operating

* Of course it is impossible to have *real* concurrency on a single-processor system. By true concurrency, we mean interleaved execution with rapid involuntary and unpredictable context switches.

0038–0644/88/050485–07$05.00

Received 1 August 1986
Revised 13 March and 15 December 1987

system (on both VAX and SUN computers), and was also ported to run stand-alone on a Motorola 68020 development system.

A number of packages have been implemented that superficially resemble the routines described here. For example, Binding[8] describes the implementation of a simple kernel providing concurrency for C. This implementation, like the others of which we are aware, differs from ours in that it does not interleave the execution of the concurrent processes — context switching occurs only upon the voluntary execution of a special primitive. This restriction makes such packages unsuitable for our purposes, as illustrated by the examples below.

PRIMITIVES

The micro-kernel has four primitives: emit, absorb, p and v. emit has the form

```
pid = emit(proceudre, stacksize, parameters)
```

where procedure is the C procedure to be executed concurrently, stacksize is the size of stack to be allocated for the procedure, parameters is the list of (zero or more) parameters to be passed to procedure, and pid is an integer result identifying the micro-process created by emit.

absorb, which waits for completion of a micro-process, has the form

```
result = absort (pid)
```

where pid is a micro-process identifier returned from emit, and result is the value returned by the procedure specified in the corresponding emit. These primitives are the same as Mesa's fork and join,[9] but we wished to avoid a name conflict with the Unix facility fork. Synchronization is provided by semaphores [cf. Reference 10, pp. 340–344]: a semaphore is represented as an integer variable s and the operations are p(&s) and v(&s).

USING THE PRIMITIVES

Because the micro-processes execute in a common address space, communication can be done using shared variables.* It is easy to construct examples in which using such variables is unsafe, e.g.

```
main () {
  int x = 0;  /* shared variable */
  int p1 = emit( count, 1000, &x );
  int p2 = emit( count, 1000, &x );
  absorb( p1 );
  absorb( p2 );
  printf("final count: %d",x);
  }
count(y)
  int *y;
{
  int i;
  for (i=0;i<10000;++i) *y = *y + 1;
  }
```

The results of running this program five times were: 14634, 16964, 16415, 15686 and 13757. In a course, programs like this were written to demonstrate and test locking protocols like the one due to Dekker [cf. Reference 10, p. 363]. Of course, it is more normal to protect a shared variable like x using a sempahore:

```
main(){
  int x = 0;
  int sema = 1;
  int p1 = emit( count, 1000, &x, &sema );
  int p2 = emit( count, 1000, &x, &sema );
  absorb( p1 );
  absorb( p2 );
  printf("final count: %d", x);
  }
count(y,s)
  int *y, *s;
{
  int i;
  for (i=0;i<10000;++i) {
    p(s);
    *y = *y + 1;
    v(s);
    }
  }
```

The appropriate use of the semaphore s prevents the two processes from interfering with each other in updating x, and the output from the program is 20,000.

HIGHER ABSTRACTIONS

The intent is that, rather than using semaphores *per se*, the user will use them to build abstractions. It is well known that monitors can be implemented from semaphores.[11] It is equally easy to implement message passing. We may define a new type medium, and the apppropriate primitives to operate on objects of that type. For example, we may choose to implement

* In our applications, we have avoided entirely the use of external variables. Instead, each variable is local to some micro-process; this owner micro-process may choose to share the memory with others by passing its address.

talk(&medium, message)

and

message = listen(&medium)

to do synchronous (blocking) communication.* Typically, the media over which a process may communicate are passed as parameters via emit. A possible implementation of talk and listen is shown in Figure 1, and Figure 2 gives a simple program that uses talk and listen. The output from this program is 4 8 12 16 20.

It is possible to implement other communication protocols, such as blocking send-receive-reply as in Thoth:[12] one has to implement a slightly different medium. However, in Thoth, messages are sent to processes rather than to media. This requirement is met by defining a new type thoth_process that is a structure containing a medium and a micro-process id. Then we define procedures thoth_create, thoth_send, thoth_receive, thoth_reply, etc., that operate on thoth_processes and are implemented in terms of emit, absorb, p and v. The Ada environment is only slightly more complicated — each Ada process may have several entries: each entry is represented by a medium. A rendezvous is identical to a Thoth send-receive-reply (entry/call = send; accept = receive; return = reply) except that it is constrained to deal with only one message at a time. Finally, Ada's select statement is implemented as a receive that conditionally receives from one of a list of media — if no previous send has occurred, the receiver waits on a common semaphore until one does.

* In our implement, message is a single long word. This long word may in fact be a pointer to a longer message. Users are free to define their own message types.

DISTRIBUTED PROGRAMMING

Because a Unix process has its own address space, executes concurrently and can communicate with other processes only over well-defined communication links, it is a suitable implementation of a *virtual machine*. A network of such virtual machines is built and connected using *pipes* as the communication links. Within each virtual machine, a subsystem is implemented using micro-processes.

Although pipes provide the physical communication between machines, it is not desirable that user micro-processes on the machines use pipes directly. We therefore use a special micro-process at each end of a pipe to act as a driver. When a user micro-process wishes to communicate with another machine, a request is sent to the driver. The driver communicates the request to the other machine where it is acted upon by the other driver.

Like those for concurrency, the abstractions that support distributed processing may have a number

```
typedef struct { int lsem, tsem, data } medrep, *medium;

medium newmedium()
{
   medium r = (medium) malloc(sizeof(medrep));
   r->lsem = 0;
   r->tsem = 1;
   return r;
   }

talk( med, dat )
   medium med;
   int dat;
{
   p( & med->tsem );
   med->data = dat;
   v( & med->lsem );
   }

int listen( med )
   medium med;
{
   int r;
   p( & med->lsem );
   r = med->data;
   v( & med->tsem );
   return r;
   }
```

Figure 1. Implementation of talk *and* listen

```
source(out)
   medium out;
{
   int i;
   for (i=1;i<=5;++i) talk(out,i);
   }
filter(in,out)
   medium in, out;
{
   int i,x;
   for (i=1;i<=5;++i){
      x = listen(in);
      talk(out,x*2);
      }
   }
sink(in)
   medium in;
{
   int i,x;
   for (i=1;i<=5;++i){
      x = listen(in);
      printf("%d ",x);
      }
   }
main(){
   medium m1 = newmedium();
   medium m2 = newmedium();
   medium m3 = newmedium();
   int p1 = emit(source,1000,m1);
   int p2 = emit(filter,1000,m1,m2);
   int p3 = emit(filter,1000,m2,m3);
   int p4 = emit(sink,1000,m3);
   absorb(p1);
   absorb(p2);
   absorb(p3);
   absorb(p4);
   }
```

Figure 2. Use of talk *and* listen

of forms. We have found *remote procedure call* to be particularly simple.* Remote procedure call has the format

rcall(machine, procedure, parameters)

and has the semantics of executing procedure on machine. Of course, it is implemented using message passing as described above. In our implementation, all machines have an identical copy of the code image, so procedure is passed as a long-word address. We can also pass a data address, but the remotely executed procedure will not be able to access the data at that address (if it attempts to do so it will probably reference some unrelated storage on the remote machine that happens to have the same address). To follow that address, it would have to rcall back to the original machine.

The code in Figure 3 represents a simplified implementation of rcall: there are only two machines (the machine parameter is therefore omitted from rcall), and the rcalled procedure must not block. Non-blocking versions of any procedure can be implemented on top of this facility by rcalling emit. After initialization, the main procedure automatically invokes the user process first_process on machine 1.

The application program in Figure 4 prints the following results to the terminal:†

```
Hello from virtual machine 1 333333 !
Hello from virtual machine 0 444444 !
Hello from virtual machine 1 555555 !
Hello from virtual machine 1 222222 !
Hello from virtual machine 1 222222 !
Hello from virtual machine 1 222222 !
Hello from virtual machine 1 222222 !
```

IMPLEMENTATION

The essential elements in implementing the micro-kernal are context switching, interleaving, semaphores, emit and absorb, and serialization within the kernel. Each micro-process has its own descriptor and its own stack, which are at opposite ends of a storage area allocated within emit. All the process descriptors are in a doubly linked circular list. The descriptor for the active microprocess is pointed to by a static variable actpcb. Each process descriptor contains a status field, which indicates that it is new, ready, waiting or finished.

A context switch entails saving the state of one process and restoring another. This state consists of the CPU registers, including the stack pointer, the program counter and the condition code. Except for the stack pointer, the entire state is pushed on the stack of a suspended process. The stack pointer itself is stored in a field of the process descriptor. Once a process's state has been saved, another process is made active by setting the stack pointer and popping the registers from the stack. The last register popped is the program counter; with this operation the CPU resumes executing the suspended micro-process.

Interleaving is accomplished using the interval timer (an interval of between 100 μs and 10 ms is used, depending on the application). Each time a process is dispatched, the interval timer is set. When

*Our experience seems to be the opposite of others: we find message passing to be the most convenient primitive for communication within a machine, and remote procedure call the most, convenient for communication with another machine. Remote procedure call can be used to implement remote message passing.

† We say to the terminal because the two machines actually have separate output files. When the output is directed to the terminal, the two files are interleaved in the order they are generated.

```
int me;      /* virtual machine id:  0 or 1 */

#include "pipes.h"
        /* declares two pipes per machine:                          */
        /* write_rcall is connected to read_rcall on other machine  */
        /* write_result is connected to read_result on other machine*/

main(){
   int first_process();
   int rcall_handler(), rch_id;
   create_pipes();
   me = !fork();
   init_pipes();
   rch_id = emit(rcall_handler,10000);
   if (me) emit(first_process, 10000);
   absorb(rch_id);
   }
rcall(proc,nparms,first_parm){            /* at most 20 parms to proc */
   static int rcall_lock = 1;
   int parm_len = (nparms+2) * sizeof(int);
   int res;

   p(&rcall_lock);
      write(write_rcall,&parm_len,sizeof(int));
      write(write_rcall,&proc,parm_len);
      read(read_result,&res,sizeof(int));
   v(&rcall_lock);
   return res;
   }
rcall_handler(){
   typedef (*proc_var)();
   int parm_len, res;
   int a[22];
   while (read(read_rcall,&parm_len,sizeof(int)) > 0) {
      if (read(read_rcall,a,parm_len) <= 0) break;
      res = (*(proc_var)a[0])(a[2],a[3],a[4],a[5],a[6],a[7], a[8],a[9],
                              a[10],a[11],a[12],a[13],a[14],a[15],a[16],
                              a[17], a[18],a[19],a[20],a[21]);
      write(write_result,&res,sizeof(int));
      }
   }
```

Figure 3. Implementation of rcall

```
hi(p){
   printf("Hello from virtual machine %d %d !\n",me,p);
   }

blab(){
   int i;
   for (i=0; i<4; ++i) rcall(hi,1,222222);
   }

first_process(){
   int res;
   int emit(), absorb();
   hi(333333);
   rcall(hi,1,444444);
   rcall(rcall,3,hi,1,555555);
   res = rcall(emit,2,blab,4000);
   rcall(absorb,1,res);
   exit(0);
   }
```

Figure 4. Use of rcall

the timer expires, an interrupt occurs, and a context switch is done. Unfortunately, the interrupt handling facility of Unix was not designed with this operation in mind — from within the handler it is not possible to examine or alter the complete state of the program. It is, however, possible to examine and alter the program counter — we save its value in a static variable and reset it to the address of an exit routine. This exit routine saves the state on the stack and stores the stack pointer in the process descriptor (pointed to by actpcb). Then control is transferred to the dispatcher. The dispatcher searches the list of descriptors to find one whose status is ready or new. If a ready process is found, its state is restored and the interval timer reset. If a new process is found, the new procedure is called and the interval timer reset. If no ready or new process is found, the system is deadlocked.

A semaphore is a long word. If no process is waiting on the semaphore, it is a simple integer value. If there are processes waiting, the semaphore is the head of a linked list of waiting processes. Each process descriptor has a link field dedicated to this application, so no storage allocation is done to construct these wait lists. The operation p decrements the value of the semaphore if it is not already 0 or the head of a wait list. If it is, the process status is set to waiting, the descriptor is added to the wait list and the code is executed to do a context switch. The operation v increments the value of the semaphore if it is not the head of a wait list. If it is, the first process descriptor is marked ready and is removed from the list.

Emit allocates storage for the stack and process descriptor. It places the parameters on the stack, followed by the address of the procedure. The new process descriptor has the status new. Emit detects the first time it is called by examining a static flag and starts the interval timer on the first call. Absorb waits for completion of a particular process. There is a special semaphore in each descriptor for this purpose. When the process finishes, it stores its result in a field of its descriptor and signals the semaphore. Once the absorbing process awakes, it copies the result from the descriptor and frees the storage of the dead process. Between completing and being absorbed, the process has the status finished.

It is necessary to ensure that the various kernel operations do not interfere with each other. Traditionally, this is ensured by locking each of the kernel operations. It is possible to disable timer interrupts by calling the procedure sigblock, but we avoided this facility altogether for three reasons: it is cumbersome to block interrupts on entry to each routine and unblock on exit, especially as a context switch may take place. Blocking and unblocking interrupts is expensive — it increases the cost of a trivial semaphore operation by two orders of magnitude. Finally, our code to perform context switching is not atomic: there is no safe point at which to unblock interrupts. We ensure the integrity of the kernel using two very simple means. First, the kernel routines are bracketed by the labels MONITOR and ENDMONITOR. When the interrupt handler is entered, the program counter is compared against these labels. If the value is between MONITOR and ENDMONITOR, the timer is reset and the handler returns immediately without altering the state. This mechanism entails no execution overhead for routines like v, and yet is perfectly safe. Sometimes the kernel must make calls to library routines that are not within the labels. These calls are all bracketed by the statements

```
CRIT = 1;
```

and

```
CRIT = 0;
```

The handler checks the static variable CRIT, and if it has the value 1, returns immediately. These two mechanisms are used exclusively to ensure the atomicity of kernel operations. It is the user's responsibility to serialize calls to non-kernel routines, including those in the C library.

MEASUREMENTS AND CONCLUSIONS

The micro-kernel is small and efficient. Its source is 200 lines — 150 lines of general C code and 50 lines of system-dependent macro definitions. On a VAX 785, the emit/absorb pair consumes 310μs. A trivial p/v pair consumes 41 μs, and a p/v pair involving a context switch consumes 110 μs. On a SUN-3/160, these figures are 470 μs, 14 μs and 50 μs, respectively. The overhead of interleaving ranges from negligible to 25 per cent depending on the timer interval.

The VAX version has been used in a number of applications. The most intensive has been to teach an advanced course in concurrent and distributed programming that has been taken by several hundred students. The major assignments in the course have been to build abstractions like those described here. The micro-kernel has stood up well under this load.

The micro-kernel was also used as a very small operating system on our Motorola 68020 development system. This version differs from the Unix SUN version only in that the system calls to manipulate the timer and to handle interrupts are replaced by the appropriate hardware primitives. Using the micro-kernel we were able to use multiple processes in writing software to exercise the system.

It can be very easy to build a concurrent programming system. The primitives can be simple and more complex ones can be built from them. We think this approach is preferable to starting with over-complex primitives and trying to force them to do what is desired.

ACKNOWLEDGEMENT

This work is supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.
2. D. M. Ritchie and K. Thompson, 'The UNIX time-sharing system', *Commun. ACM*, **17**, (7), 365–375 (1974).
3. S. Leffler, W. Joy and M. McKusick, *Unix Programmer's Manual, 4.2 Berkeley Software Distribution*, University of California, Berkeley, 1983.
4. *O.S. PL/I Checkout and Optimizing Compilers: Language Reference Manual*, IBM GC33–0009.
5. *Reference Manual for the Programming Language Ada*, U.S. Department of Defense, ANSI/MIL-STD-1815-A, 1983.
6. N. Gehani and W. Roome, 'Concurrent C', *Software — Practice and Experience*, **16**, 821–844 (1986).
7. R. C. Holt, *Concurrent Euclid, the Unix System, and Tunis*, Addison Wesley, 1983.
8. C. Binding, 'Cheap concurrency in C', *ACM SIGPLAN Notices*, **20**, (9), 21–26 (1985).
9. J. Mitchell, W. Maybury and R. Sweet, *Mesa Language Manual*, Xerox, Palo Alto, 1979.
10. J. Peterson and A. Silberschatz, *Operating Systems Concepts*, Addison Wesley, 1985.
11. C. A. R. Hoare, 'Monitors: an operating system structuring concept', *Commun. ACM*, **17**, (10), 549–557, (1974).
12. D. Cheriton, M. A. Malcolm, L. S. Melen and G. R. Sager, 'Thoth, a portable real-time operating system', *Commun. ACM*, **22**, (2), 105–115 (1979).